Adjoint methods in computational finance

Mike Giles

Mathematical and Computational Finance Group,
Mathematical Institute, University of Oxford

Oxford-Man Institute of Quantitative Finance

Global Derivatives USA

Nov 22, 2013

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

Answer 1: no, in theory, and also in practice if A, B, C are square

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

Answer 1: no, in theory, and also in practice if A, B, C are square

Answer 2: yes, in practice, if A, B, C have dimensions 1×10^5, $10^5\times 10^5$, $10^5\times 10^5$.

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

A question!

Given compatible matrices A, B, C does it matter how one computes the product $A\,B\,C$? (i.e. $(A\,B)\,C$ or $A\,(B\,C)$?)

Answer 1: no, in theory, and also in practice if A, B, C are square

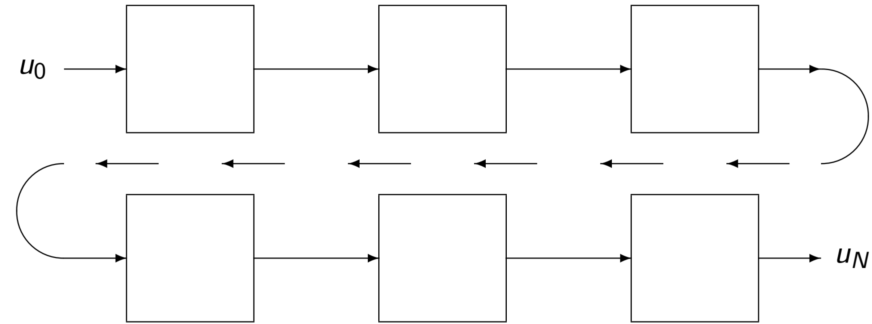
Answer 2: yes, in practice, if A, B, C have dimensions 1×10^5, $10^5\times 10^5$, $10^5\times 10^5$.

$$\left(\cdot \;\cdot \;\cdot \;\cdot \;\cdot \right) \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Point: this is all about computational efficiency

Generic black-box problem

An input vector u_0 leads to a scalar output u_N:



Each box could be a mathematical step (calibration, spline, pricing) or a computer code, or one computer instruction

Key assumption: each step is (locally) differentiable

Generic black-box problem

Let $\dot{u}_n$ represent the derivative of u_n with respect to one particular element of input u_0. Differentiating black-box processes gives

$$\dot{u}_{n+1} = D_n\, \dot{u}_n, \qquad D_n \equiv \frac{\partial u_{n+1}}{\partial u_n}$$

and hence

$$\dot{u}_N = D_{N-1}\, D_{N-2}\, \ldots\, D_1\, D_0\, \dot{u}_0$$

- standard "forward mode" approach multiplies matrices from right to left – very natural
- each element of u_0 requires its own sensitivity calculation – cost proportional to number of inputs

Generic black-box problem

Let $\overline{u}_n$ be the derivative of output u_N with respect to u_n.

$$\overline{u}_n \equiv \left(\frac{\partial u_N}{\partial u_n}\right)^T = \left(\frac{\partial u_N}{\partial u_{n+1}}\;\frac{\partial u_{n+1}}{\partial u_n}\right)^T = D_n^T\, \overline{u}_{n+1}$$

and hence

$$\overline{u}_0 = D_0^T\, D_1^T\, \ldots\, D_{N-2}^T\, D_{N-1}^T\, \overline{u}_N$$

and $\overline{u}_N = 1$.

- $\overline{u}_0$ gives sensitivity of u_N to all elements of u_n at a fixed cost, not proportional to the size of u_0
- a different output would require a separate adjoint calculation – cost proportional to number of outputs

Generic black-box problem

It looks easy (?) – what's the catch?

- need to do original nonlinear calculation to compute and store D_n (or at least enough data to compute $D_n^T \overline{u}_{n+1}$) before doing adjoint reverse pass – storage requirements can be significant for PDEs
- practical implementation can be tedious if hand-coded – use automatic differentiation tools
- need care in treating black-boxes which involve a fixed point iteration
- derivative may not be as accurate as original approximation

Automatic differentiation

We now consider a single black-box component, which is actually the outcome of a computer program.

A computer instruction creates an additional new value:

$$\mathbf{u}_{n+1} \;=\; \mathbf{f}_n(\mathbf{u}_n) \;\equiv\; \begin{pmatrix} \mathbf{u}_n \\ f_n(\mathbf{u}_n) \end{pmatrix}$$

A computer program is the composition of N such steps:

$$\mathbf{u}_N = \mathbf{f}_{N-1} \circ \mathbf{f}_{N-2} \circ \ldots \circ \mathbf{f}_1 \circ \mathbf{f}_0(\mathbf{u}_0)$$

and the final output of interest is the last element of $\mathbf{u}_N$.

Automatic differentiation

In forward mode, differentiation gives

$$\dot{\mathbf{u}}_{n+1} = D_n\, \dot{\mathbf{u}}_n, \qquad D_n \equiv \begin{pmatrix} I_n \\ \partial f_n / \partial \mathbf{u}_n \end{pmatrix},$$

and hence

$$\dot{\mathbf{u}}_N \;=\; D_{N-1}\; D_{N-2} \ldots\; D_1\; D_0\; \dot{\mathbf{u}}_0.$$

This is relatively intuitive – but the next step is not.

Automatic differentiation

In reverse mode, we have

$$\overline{\mathbf{u}}_n = \Big(D_n\Big)^T\; \overline{\mathbf{u}}_{n+1}.$$

and hence

$$\overline{\mathbf{u}}_0 = (D_0)^T\; (D_1)^T \ldots\; (D_{N-2})^T\; (D_{N-1})^T\; \overline{\mathbf{u}}_N$$

where the last element of $\overline{\mathbf{u}}_N$ is 1, and the rest are zero.

Note: need to go forward through original calculation to compute/store the D_n, then go in reverse to compute $\overline{\mathbf{u}}_n$

Automatic differentiation

At the level of a single instruction

$$c = f(a, b)$$

the forward mode is

$$\begin{pmatrix} \dot{a} \\ \dot{b} \\ \dot{c} \end{pmatrix}_{n+1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ \frac{\partial f}{\partial a} & \frac{\partial f}{\partial b} \end{pmatrix} \begin{pmatrix} \dot{a} \\ \dot{b} \end{pmatrix}_n$$

and so the reverse mode is

$$\begin{pmatrix} \overline{a} \\ \overline{b} \end{pmatrix}_n = \begin{pmatrix} 1 & 0 & \frac{\partial f}{\partial a} \\ 0 & 1 & \frac{\partial f}{\partial b} \end{pmatrix} \begin{pmatrix} \overline{a} \\ \overline{b} \\ \overline{c} \end{pmatrix}_{n+1}$$

Automatic differentiation

This gives a prescriptive algorithm for reverse mode differentiation.

$$\overline{a}_n = \overline{a}_{n+1} + \frac{\partial f}{\partial a}\,\overline{c}_{n+1} \quad \longrightarrow \quad \overline{a} \mathrel{+}= \frac{\partial f}{\partial a}\,\overline{c}$$

$$\overline{b}_n = \overline{b}_{n+1} + \frac{\partial f}{\partial b}\,\overline{c}_{n+1} \quad \longrightarrow \quad \overline{b} \mathrel{+}= \frac{\partial f}{\partial b}\,\overline{c}$$

mathematics computer program

Key observation: 1 multiply in original nonlinear calculation turns into 2 multiply-add operations in reverse mode, so at most the reverse mode costs only a factor 4 more than the original calculation.

(This ignores the cost of memory references.)

Automatic differentiation

A simple example with inputs a, b, c

$$\begin{aligned} d &:= a + b \\ e &:= c\, d \\ f &:= \exp(a) \\ g &:= e + f \end{aligned}$$

Automatic differentiation

A simple example with inputs a, b, c and $\dot{a}, \dot{b}, \dot{c}$

$$\begin{aligned} \dot{d} &:= \dot{a} + \dot{b} \\ d &:= a + b \\ \dot{e} &:= \dot{c}\, d + c\, \dot{d} \\ e &:= c\, d \\ \dot{f} &:= \exp(a)\, \dot{a} \\ f &:= \exp(a) \\ \dot{g} &:= \dot{e} + \dot{f} \\ g &:= e + f \end{aligned}$$

Automatic differentiation

In reverse mode, we get

$$
\begin{aligned}
d &:= a + b \\
e &:= c\, d \\
f &:= \exp(a) \\
g &:= e + f
\end{aligned}
$$

$$
\begin{aligned}
\overline{e} &\mathrel{+}= \overline{g} \\
\overline{f} &\mathrel{+}= \overline{g} \\
\overline{a} &\mathrel{+}= \exp(a)\ \overline{f} \\
\overline{c} &\mathrel{+}= d\ \overline{e} \\
\overline{d} &\mathrel{+}= c\ \overline{e} \\
\overline{a} &\mathrel{+}= \overline{d} \\
\overline{b} &\mathrel{+}= \overline{d}
\end{aligned}
$$

Automatic differentiation tools

Manual implementation is possible but can be very tedious,
so automated tools have been developed, following two approaches:

- operator overloading (dco, ADOL-C, FADBAD++)
- source code transformation (Tapenade, TAF/TAC++)

Operator overloading

In forward mode, define a new datatype for value x and sensitivity $\dot{x}$.

Then define operators accordingly:

$$
\begin{pmatrix} x \\ \dot{x} \end{pmatrix} + \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x+y \\ \dot{x}+\dot{y} \end{pmatrix}
$$

$$
\begin{pmatrix} x \\ \dot{x} \end{pmatrix} * \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x\, y \\ y\, \dot{x} + x\, \dot{y} \end{pmatrix}
$$

$$
\begin{pmatrix} x \\ \dot{x} \end{pmatrix} / \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x/y \\ (1/y)\, \dot{x} - (x/y^2)\, \dot{y} \end{pmatrix}
$$

$$
\exp \begin{pmatrix} x \\ \dot{x} \end{pmatrix} = \begin{pmatrix} \exp(x) \\ \exp(x)\, \dot{x} \end{pmatrix}
$$

Note: this works as well if the sensitivity $\dot{x}$ is a vector so we compute several different sensitivities at the same time.

Operator overloading

Operator overloading doesn't seem to extend naturally to reverse mode.

What is done is to make a record (referred to as a "tape") of all operations performed in the original calculation, and their input operands.

Then, it is possible to work backwards through the tape performing the necessary adjoint calculations.

The performance depends strongly on the implementation – a lot of data is stored and then brought back.

Also very useful for code validation.

Source code transformation

- programmer supplies black-box code which takes u as input and produces $v=f(u)$ as output
- in forward mode, AD tool generates new code which takes u and $\dot{u}$ as input, and produces v and $\dot{v}$ as output

$$\dot{v} = \left(\frac{\partial f}{\partial u}\right)\ \dot{u}$$

- in reverse mode, AD tool generates new code which takes u and $\overline{v}$ as input, and produces v and $\overline{u}$ as output

$$\overline{u} = \left(\frac{\partial f}{\partial u}\right)^T \overline{v}$$

Source code transformation

Note that for any choice of $\dot{u}$ and $\overline{v}$,

$$\overline{v}^T\ \dot{v} \quad = \quad \overline{v}^T \left(\frac{\partial f}{\partial u}\right)\ \dot{u} \quad = \quad \overline{u}^T\ \dot{u}$$

The left hand side comes from the forward mode code which computes $\dot{v}$; the right hand side comes from the reverse mode code which computes $\overline{u}$.

Checking this equality is an important validation step when developing forward and reverse mode sensitivity code, especially when it is done by hand, not through automatic differentation.

Fixed point iteration

Suppose a black-box computes output v from input u by solving the nonlinear equations

$$f(u,v)=0$$

using the fixed-point iteration

$$v_{n+1} = v_n - P(u,v_n)\ f(u,v_n)$$

For a Newton iteration, P is the inverse Jacobian, but P could also correspond to a multigrid cycle in an iterative solver – this is relevant to multi-dimensional finite difference methods.

Fixed point iteration

A naive forward mode differentiation uses the fixed-point iteration

$$\dot{v}_{n+1} = \dot{v}_n - \left(\frac{\partial P}{\partial u}\,\dot{u} + \frac{\partial P}{\partial v}\,\dot{v}_n\right)\ f(u,v_n) - P(u,v_n)\left(\frac{\partial f}{\partial u}\,\dot{u} + \frac{\partial f}{\partial v}\,\dot{v}_n\right)$$

but it is more efficient to use

$$\dot{v}_{n+1} = \dot{v}_n - P(u,v)\ \left(\frac{\partial f}{\partial u}\,\dot{u} + \frac{\partial f}{\partial v}\,\dot{v}_n\right)$$

to iteratively solve

$$\frac{\partial f}{\partial u}\,\dot{u} + \frac{\partial f}{\partial v}\,\dot{v} = 0$$

Fixed point iteration

Since

$$\dot{v} = -\left(\frac{\partial f}{\partial v}\right)^{-1} \frac{\partial f}{\partial u}\,\dot{u}$$

the adjoint is

$$\overline{u} = -\left(\frac{\partial f}{\partial u}\right)^{T} \left(\left(\frac{\partial f}{\partial v}\right)^{T}\right)^{-1} \overline{v} = \left(\frac{\partial f}{\partial u}\right)^{T} \overline{w}$$

where

$$\left(\frac{\partial f}{\partial v}\right)^{T} \overline{w} + \overline{v} = 0$$

Fixed point iteration

This can be solved iteratively using

$$\overline{w}_{n+1} = \overline{w}_n - P^T(u,v)\left(\left(\frac{\partial f}{\partial v}\right)^{T} \overline{w}_n + \overline{v}\right)$$

and this is guaranteed to converge (well!) since

$$P^T(u,v)\left(\frac{\partial f}{\partial v}\right)^{T}$$

has the same eigenvalues as

$$P(u,v)\,\frac{\partial f}{\partial v}$$

Final comments

- forward mode sensitivity analysis is just classic linear sensitivity analysis – very intuitive
- reverse mode adjoint sensitivity analysis computes exactly the same value, but does so much more efficiently when you want the sensitivity of one output to many inputs
- the adjoint approach is not intuitive, so don't worry if it seems strange – trust the mathematics, and proceed slowly

Adjoint methods in computational finance

Mike Giles

Mathematical and Computational Finance Group,
Mathematical Institute, University of Oxford

Oxford-Man Institute of Quantitative Finance

Global Derivatives USA

Nov 22, 2013

SDEs – the basics

- LRM and pathwise sensitivity approaches
- adjoint pathwise approach
- use of automatic differentiation software
- path dependent options

Monte Carlo simulation

If we want the expected value of a payoff function P which depends on an underlying quantity S, so that

$$V = \mathbb{E}[P(S)] = \int P(S)\, p_S(\theta; S)\, \mathrm{d}S$$

where p_S is the probability distribution for S which depends on an input parameter θ, then the Monte Carlo estimate is

$$M^{-1} \sum_{m=1}^{M} P(S^{(m)})$$

where the samples $S^{(m)}$ are generated independently.

LRM sensitivity analysis

If p_S is a differentiable function of θ, then

$$\frac{\partial V}{\partial \theta} = \int P(S)\, \frac{\partial p_S}{\partial \theta}\, \mathrm{d}S = \int P(S)\, \frac{\partial \log p_S}{\partial \theta}\, p_S(S)\, \mathrm{d}S$$

which is equivalent to

$$\frac{\partial V}{\partial \theta} = \mathbb{E}\left[P(S)\, \frac{\partial \log p_S}{\partial \theta} \right]$$

which can be estimated as

$$M^{-1} \sum_{m=1}^{M} P(S^{(m)})\, \frac{\partial \log p_S^{(m)}}{\partial \theta}$$

LRM sensitivity analysis

This is the Likelihood Ratio Method for computing sensitivities.

Its great strength is that there are no restrictions on $P(S)$ (e.g. it can be discontinuous) but it has two big drawbacks:

- it requires a known distribution p_S (e.g. log-normal)
- the Monte Carlo estimator usually has a much larger variance than alternative methods

Monte Carlo simulation

Alternatively, suppose S depends on θ and a simple random variable Z which does not depend on θ (e.g. multivariate Normal).

Then we have

$$V \;=\; \mathbb{E}[P(\,S(\theta;Z)\,)] \;=\; \int P(\,S(\theta;Z)\,)\; p_Z(Z)\;\mathrm{d}Z$$

and the Monte Carlo estimate remains

$$M^{-1}\sum_{m=1}^{M} P(\,S(Z^{(m)})\,)$$

where the samples $Z^{(m)}$ are generated independently.

Pathwise sensitivity analysis

If P is a differentiable function of θ, then

$$\frac{\partial V}{\partial \theta} \;=\; \int \frac{\partial P}{\partial S}\,\frac{\partial S}{\partial \theta}\; p_Z(Z)\;\mathrm{d}Z$$

which is equivalent to

$$\frac{\partial V}{\partial \theta} \;=\; \mathbb{E}\left[\frac{\partial P}{\partial S}\,\frac{\partial S}{\partial \theta}\right]$$

This is also OK if P is continuous and piecewise differentiable, but not if P is discontinuous.

For example, for a simple digital option with value 0 or 1 depending on S then locally $\dfrac{\partial P}{\partial S}=0$, but $\dfrac{\partial V}{\partial \theta}\neq 0$.

Pathwise sensitivity analysis

This gives the pathwise sensitivity estimate

$$M^{-1}\sum_{m=1}^{M} \frac{\partial P}{\partial S}\;\dot{S}^{(m)}$$

where $\dot{S}$ is the sensitivity keeping fixed all of the random numbers.

Note that this corresponds very naturally to applying the forward mode sensitivity analysis to the standard Monte Carlo estimator, but my derivation shows the need for $P(S)$ to be continuous – this is its big weakness.

Monte Carlo sensitivities

For European payoff $P(S_N)$, the forward mode sensitivity calculation is:

- set $S_0(\theta), \dot{S}_0$
- for timestep n from 0 to $N-1$:

  compute random numbers Z_n

  compute $S_{n+1} = f_n(\theta; S_n, Z_n)$

  compute $\dot{S}_{n+1} = \dfrac{\partial f_n}{\partial S_n}\, \dot{S}_n + \dfrac{\partial f_n}{\partial \theta}$
- calculate payoff $P(\theta; S_N)$ and $\dot{P} = \dfrac{\partial P}{\partial S_N}\, \dot{S}_N + \dfrac{\partial P}{\partial \theta}$

Monte Carlo sensitivities

The corresponding adjoint (reverse mode) sensitivity is

$$M^{-1} \sum_{m=1}^{M} \overline{\theta}^{(m)}$$

where $\overline{\theta}^{(m)}$ corresponds to $\left(\dfrac{\partial P}{\partial \theta}\right)^T$ for m^{th} path

Note: the adjoint sensitivity is the same as the standard pathwise sensitivity, so it is valid under the same conditions (e.g. $P(S)$ Lipschitz and piecewise differentiable)

Monte Carlo sensitivities

For European payoff $P(S_N)$, the adjoint calculation involves:

- set $S_0(\theta)$
- for timestep n from 0 to $N-1$:

  compute and store random numbers Z_n

  compute and store $S_{n+1} = f_n(\theta; S_n, Z_n)$
- calculate payoff $P(\theta; S_N)$

- set $\overline{S}_N = \dfrac{\partial P}{\partial S_N}$ and $\overline{\theta} = \dfrac{\partial P}{\partial \theta}$
- for timestep n from $N-1$ to 0

  compute $\overline{S}_n$ and $\overline{\theta}$ increment
- add final $\overline{\theta}$ increment $\overline{S}_0^{\,T} \dot{S}_0$

Trivial example

Geometric Brownian Motion SDE, European call payoff, Euler discretisation:

Given S_0, r, σ, T, K, then

$$S_{n+1} = S_n + r\, S_n\, \Delta t + \sigma\, S_n \sqrt{\Delta t}\, Z_n \quad n = 0, \ldots N-1$$

and the payoff is

$$P = \exp(-rT)\ \max(0, S_N - K)$$

Note that there are 5 different sensitivities which can be computed here.

Trivial example

In the forward mode, given $\dot{S}_0, \dot{r}, \dot{\sigma}, \dot{T}, \dot{K}$, then

$$\begin{aligned}\dot{S}_{n+1} \;=\; & \left(1 + r\,\Delta t + \sigma\,\sqrt{\Delta t}\,Z_n\right)\dot{S}_n \\ & + S_n\,\Delta t\;\dot{r} \\ & + S_n\,\sqrt{\Delta t}\,Z_n\;\dot{\sigma} \\ & + \left(r\,S_n + \tfrac{1}{2}\,\sigma\,S_n\,\Delta t^{-1/2}\,Z_n\right)N^{-1}\,\dot{T}\end{aligned}$$

and the payoff sensitivity is

$$\dot{P} = \exp(-rT)\,\mathbf{1}_{S_N>K}\,\left(\dot{S}_N - \dot{K} - (S_N - K)\,(T\dot{r} + r\dot{T})\right)$$

Trivial example

To obtain the sensitivities to all of S_0, r, σ, T, K, in one pass, we can set

$$\dot{S}_0 = \begin{pmatrix}1\\0\\0\\0\\0\end{pmatrix},\; \dot{r} = \begin{pmatrix}0\\1\\0\\0\\0\end{pmatrix},\; \dot{\sigma} = \begin{pmatrix}0\\0\\1\\0\\0\end{pmatrix},\; \dot{T} = \begin{pmatrix}0\\0\\0\\1\\0\end{pmatrix},\; \dot{K} = \begin{pmatrix}0\\0\\0\\0\\1\end{pmatrix},$$

and then the final $\dot{P}$ will be a vector of sensitivities:

$$\dot{P} = \begin{pmatrix}\frac{\partial P}{\partial S_0}\\ \frac{\partial P}{\partial r}\\ \frac{\partial P}{\partial \sigma}\\ \frac{\partial P}{\partial T}\\ \frac{\partial P}{\partial K}\end{pmatrix}$$

Trivial example

In the reverse mode, we start with

$$\begin{aligned}\overline{S}_N &= \exp(-rT)\,\mathbf{1}_{S_N>K}\\ \overline{r} &= -\exp(-rT)\,\mathbf{1}_{S_N>K}\,(S_N - K)\,T\\ \overline{\sigma} &= 0\\ \overline{T} &= -\exp(-rT)\,\mathbf{1}_{S_N>K}\,(S_N - K)\,r\\ \overline{K} &= -\exp(-rT)\,\mathbf{1}_{S_N>K}\end{aligned}$$

and then loop over timesteps from $N-1$ to 0 using

$$\begin{aligned}\overline{S}_n &= \left(1 + r\,\Delta t + \sigma\,\sqrt{\Delta t}\,Z_n\right)\overline{S}_{n+1}\\ \overline{r} &\mathrel{+}= S_n\,\Delta t\;\overline{S}_{n+1}\\ \overline{\sigma} &\mathrel{+}= S_n\,\sqrt{\Delta t}\,Z_n\;\overline{S}_{n+1}\\ \overline{T} &\mathrel{+}= \left(r\,S_n + \tfrac{1}{2}\,\sigma\,S_n\,\Delta t^{-1/2}\,Z_n\right)N^{-1}\;\overline{S}_{n+1}\end{aligned}$$

to get all 5 sensitivities, $\overline{S}_0$, $\overline{r}$, $\overline{\sigma}$, $\overline{T}$, $\overline{K}$

Monte Carlo sensitivities

In more complicated cases, AD tools can simplify the software development process, especially for the reverse mode.

If a routine

```
step(n,theta,S,Z)
```

performs the nth timestep calculation, taking θ, S_n, Z_n as input and returning S_{n+1}, then AD tools can generate a routine

```
step_b(n,theta,theta_b,S,S_b,Z)
```

which takes inputs $\theta, \overline{\theta}, S_n, \overline{S}_{n+1}, Z_n$ and returns $\overline{S}_n$ and an updated $\overline{\theta}$.

AD tools can also generate the adjoint routines for the S_0 initialisation and the payoff evaluation.

Monte Carlo sensitivities

Some more implementation detail:

- first, go forward through the path storing the state S_n at each timestep (corresponds to "checkpointing" in AD terminology)
- then, go backwards through the path, using reverse mode AD for each step – this will re-do the internal calculations for the timestep and then do its adjoint
- when hand-coding for maximum performance, I also store the result of any very expensive operations (typically exp) to avoid having to re-do these

Note that this is different from applying AD to the entire path, which would require a lot of storage – it's often cheaper to re-calculate the internal variables rather than fetch them from main memory

Monte Carlo sensitivities

By computing $\overline{\theta} \equiv \dfrac{\partial P}{\partial \theta}$ for each path individually, we can also compute a confidence interval for the sensitivity estimate.

If we had applied AD to the entire code which computes:

- the estimate
- the confidence interval

then we would have obtained:

- the sensitivity of the estimate
- the sensitivity of the confidence interval, not the confidence interval of the sensitivity – a subtle but important difference.

LIBOR Market Model

(*Smoking Adjoints*: 2006 RISK article with Paul Glasserman)

As an example, consider the LIBOR market model of BGM, with $m{+}1$ bond maturities T_i, with spacings $T_{i+1} - T_i = \delta$.

The forward rate for the interval $[T_i, T_{i+1})$ satisfies

$$\frac{\mathrm{d}L_i(t)}{L_i(t)} = \mu_i(L(t))\,\mathrm{d}t + \sigma_i^\top\,\mathrm{d}W(t), \quad 0 \le t \le T_i,$$

where $$\mu_i(L(t)) = \sum_{j=\eta(t)}^{i} \frac{\sigma_i^\top \sigma_j\, \delta L_j(t)}{1 + \delta L_j(t)},$$

and $\eta(t)$ is the index of the next maturity date.

For simplicity, we keep $L_i(t)$ constant after maturity, and take the volatilities to be a function of time to maturity, $\sigma_i(t) = \sigma_{i-\eta(t)+1}(0)$.

LIBOR Market Model

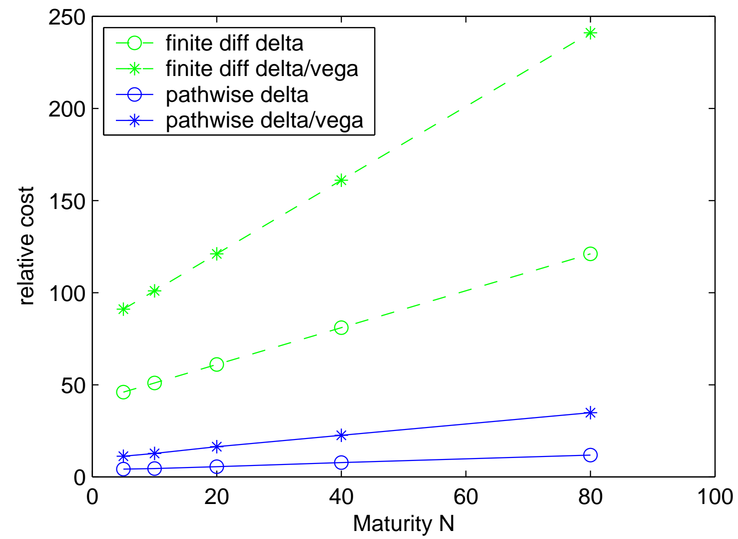
The LMM portfolio has 15 swaptions all expiring at the same time, N periods in the future, involving payments/rates over an additional 40 periods in the future.

Interested in computing Deltas, sensitivity to initial $N{+}40$ forward rates, and Vegas, sensitivity to initial $N{+}40$ volatilities.

Focus is on the cost of calculating the portfolio value and the sensitivities, relative to just the value.

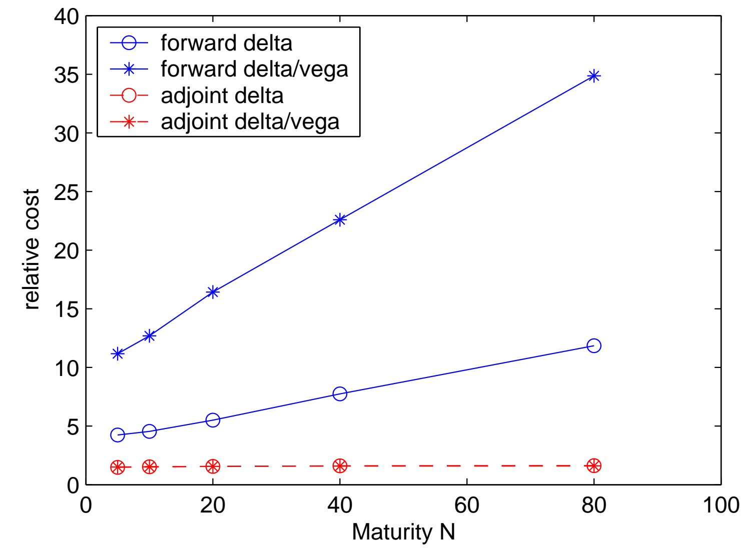
LIBOR Market Model

Finite differences versus forward pathwise sensitivities:



LIBOR Market Model

Forward versus adjoint pathwise sensitivities:



Adjoint methods in computational finance

Mike Giles

Mathematical and Computational Finance Group,
Mathematical Institute, University of Oxford

Oxford-Man Institute of Quantitative Finance

Global Derivatives USA

Nov 22, 2013

Additional Monte Carlo complications

- path-dependent payoffs
- efficiency improvement for handling multiple European payoffs (Christoph Kaebe & Ekkehard Sachs)
- binning for expensive pre-processing steps (Luca Capriotti)
- interpolation from a local vol surface
- handling discontinuous payoffs
- second order Greeks
- American and Bermudan options

Path dependent payoffs

The simplest way to handle path dependent payoffs is to make the payoff depend only on the final state S_N by including one or more of the following as part of a larger state $\widehat{S}_n$

- $\sum_{m\leq n} S_m$ for Asian options
- $\min_{m\leq n} S_m, \ \max_{m\leq n} S_m$ for lookback options
- sum of all (discounted) cash payments made so far

We then have an augmented timestep calculation

$$\widehat{S}_{n+1} = \widehat{f}_n(\theta; \widehat{S}_n, Z_n)$$

and can again use AD tools to generate the whole adjoint code.

Multiple European payoffs

Suppose that you have

- n_θ input parameters
- n_P different payoffs
- dimension d path simulation

If n_θ is smallest, use forward mode sensitivity analysis

If n_P is smallest, use reverse mode sensitivity analysis

What if d is smallest?

Multiple European payoffs

Going back to the original matrix question, what is the best way of computing this?

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Multiple European payoffs

The most efficient approach is

- perform d adjoint calculations to determine

$$\frac{\partial S_N}{\partial \theta}$$

- perform d forward sensitivity calculations to determine

$$\frac{\partial P_k}{\partial S_N}$$

- combine these to obtain

$$\frac{\partial P_k}{\partial \theta} = \frac{\partial P_k}{\partial S_N} \, \frac{\partial S_N}{\partial \theta}$$

Binning

The need for binning is best demonstrated by the case of correlation Greeks – computing the sensitivity of the option value to each element in the correlation matrix.

Given correlation matrix Ω, Cholesky factor L is lower-triangular matrix such that $L\,L^T = \Omega$. If Z is a vector of uncorrelated unit Normals, then $L\,Z$ has variance Ω.

The standard pricing calculation has three stages

- perform Cholesky factorisation
- do M path calculations
- compute average and confidence interval

How do we compute the adjoint sensitivity to the correlation coefficients?

Binning

If we apply the reverse mode AD approach to the entire calculation, then we get an estimate of

- the sensitivity of the price
- the sensitivity of the confidence interval,
  not the confidence interval for the sensitivity!

To get the confidence interval for the sensitivity, for each path we can do the adjoint of the Cholesky factorisation, so we compute $\overline{\theta}$ for each path and then compute an average and confidence interval in the usual way.

However, this greatly increases the computational cost.

Binning

The binning approach splits the M paths into K groups (grouped arbitrarily, unlike "binning" in some other contexts).

For each group, it uses the full AD approach to efficiently compute an estimate of the price sensitivity.

It then uses the variability between the group averages to estimate the confidence interval.

Needs

- $K \gg 1$ to get a good estimate of the confidence interval
- $K \ll M$ for cost of K adjoint Cholesky calculations to be smaller than M path calculations

Local volatility example

The same binning approach can be used for a Monte Carlo calculation using a local volatility surface:

- market implied vol $\sigma_I \implies$ local vol σ_L at a few points using Dupire's formula
- local vol σ_L at a few points $\implies$ σ'_L through cubic spline procedure
- M Monte Carlo path calculation, using spline evaluation to obtain local volatility
- compute average and confidence interval

The adjoint of the path calculation will give increments to $\overline{\sigma_L}$ and $\overline{\sigma'_L}$. Then, for each group of paths, can use adjoint of first two stages to get an estimate for the sensitivity to market implied vol data.

Local volatility example

At a particular time t, cubic spline evaluation at S is of form

$$\sigma(S) = a_j(S)\,\sigma_j + b_j(S)\,\sigma_{j+1} + c_j(S)\,\sigma'_j + d_j(S)\,\sigma'_{j+1}$$

In the forward mode we get

$$\begin{aligned}\dot\sigma(S) &= a_j(S)\,\dot\sigma_j + b_j(S)\,\dot\sigma_{j+1} + c_j(S)\,\dot\sigma'_j + d_j(S)\,\dot\sigma'_{j+1} \\ &\quad + \left(a'_j(S)\,\sigma_j + b'_j(S)\,\sigma_{j+1} + c'_j(S)\,\sigma'_j + d'_j(S)\,\sigma'_{j+1}\right)\dot S\end{aligned}$$

and in the reverse mode we get

$$\begin{aligned}\overline{\sigma}_j &\mathrel{+}= a_j(S)\,\overline{\sigma}(S) \\ \overline{\sigma}_{j+1} &\mathrel{+}= b_j(S)\,\overline{\sigma}(S) \\ \overline{\sigma}'_j &\mathrel{+}= c_j(S)\,\overline{\sigma}(S) \\ \overline{\sigma}'_{j+1} &\mathrel{+}= d_j(S)\,\overline{\sigma}(S) \\ \overline{S} &\mathrel{+}= \left(a'_j(S)\,\sigma_j + b'_j(S)\,\sigma_{j+1} + c'_j(S)\,\sigma'_j + d'_j(S)\,\sigma'_{j+1}\right)\,\overline{\sigma}(S)\end{aligned}$$

Discontinuous payoffs

The biggest limitation of the pathwise sensitivity method (both forward mode and reverse mode) is that it cannot handle discontinuous payoffs.

There are 3 main ways to deal with this:

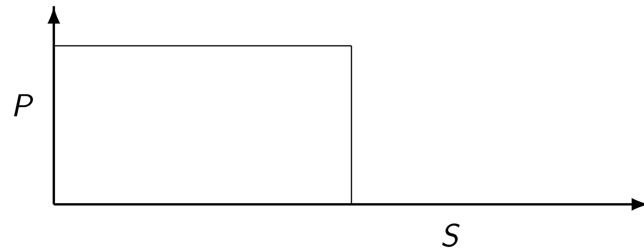
- explicitly smoothed payoffs
- using conditional expectation to smooth the payoff
- "vibrato" Monte Carlo

Of course, we can also switch to Likelihood Ratio Method or Malliavin calculus, but then I don't see how to get the efficiency benefits of adjoint methods.

Discontinuous payoffs

Explicitly-smoothed payoffs replace the discontinuous payoff by a smooth (or at least continuous) alternative.

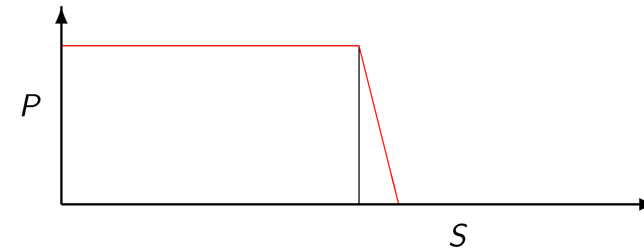
Digital options $P(S) \equiv H(S-K)$ can be replaced by a piecewise linear approximation:



Discontinuous payoffs

Explicitly-smoothed payoffs replace the discontinuous payoff by a smooth (or at least continuous) alternative.

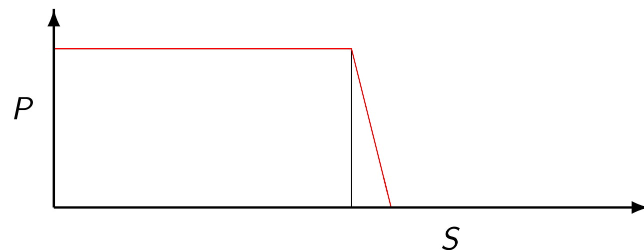
Digital options $P(S) \equiv H(S-K)$ can be replaced by a piecewise linear approximation:



Discontinuous payoffs

Explicitly-smoothed payoffs replace the discontinuous payoff by a smooth (or at least continuous) alternative.

Digital options $P(S) \equiv H(S-K)$ can be replaced by a piecewise linear approximation:



or something much smoother such as $\Phi\left(\frac{S-K}{\delta}\right)$ which has an $O(\delta^2)$ bias

Discontinuous payoffs

Implicitly-smoothed payoffs use conditional expectations.

My favourite is for barrier options, where a Brownian Bridge conditional expectation computes the probability that the path has crossed the barrier within a timestep – see Glasserman's book, *Monte Carlo Methods for Financial Engineering*, pp. 366-370.

This improves the weak convergence to first order, and also makes the payoff differentiable.

Discontinuous payoffs

With digital options, can stop the path simulation one timestep before maturity.

Conditional on the value S_{N-1}, an Euler discretisation for the final timestep has the form

$$S_N = S_{N-1} + \mu_{N-1}\,\Delta t + \sigma_{N-1}\,\Delta W_{N-1}$$

which gives a (multi-variate) Normal p.d.f. for S_N

In simple cases it is possible to analytically evaluate

$$\mathbb{E}\left[\, P(S_N)\,|\, S_{N-1}\right]$$

and this will be a smooth function of S_{N-1} so we can use the pathwise sensitivity method.

Discontinuous payoffs

Continuing this digital example, in more complicated multi-dimensional cases it is not possible to analytically evaluate the conditional expectation.

Instead, we can apply the Likelihood Ratio Method for the final step
– I called this the "vibrato" method because of the uncertainty in the final value S_N

Need to read my paper for full details for multi-dimensional applications
– I'll give an outline for a scalar SDE.

Its main weakness is that the variance is $O(\Delta t^{-1/2})$, but it is much better than the $O(\Delta t^{-1})$ variance of the standard Likelihood Ratio Method, and you get the full benefit of adjoints.

Discontinuous payoffs

Conditional on the value of S_{N-1}, the probability distribution for S_N is

$$p_s(\theta, S_{N-1}; S)$$

where p_s is a Normal distribution with a mean and variance which depend on S_{N-1}, and perhaps also directly on θ.

Hence, using the LRM approach,

$$\frac{\partial}{\partial\theta}\,\mathbb{E}\left[\, P(S_N)\,|\, S_{N-1}\right] = \mathbb{E}\left[\, P(S_N)\, \left(\frac{\partial \log p_s}{\partial\theta}\right)_{total} \,|\, S_{N-1}\right]$$

with

$$\left(\frac{\partial \log p_s}{\partial\theta}\right)_{total} = \frac{\partial \log p_s}{\partial S_{N-1}}\,\frac{\partial S_{N-1}}{\partial\theta} + \frac{\partial \log p_s}{\partial\theta}$$

The conditional expectation can be estimated by averaging over a number of samples for Z_{N-1}, the random number used for the final timestep.

Discontinuous payoffs

Re-arranging we get

$$\begin{aligned}\frac{\partial}{\partial\theta}\,\mathbb{E}\left[\, P(S_N)\,|\, S_{N-1}\right] &= \mathbb{E}\left[\, P(S_N)\,\frac{\partial \log p_s}{\partial S_{N-1}} \,|\, S_{N-1}\right]\frac{\partial S_{N-1}}{\partial\theta} \\ &+ \mathbb{E}\left[\, P(S_N)\,\frac{\partial \log p_s}{\partial\theta} \,|\, S_{N-1}\right]\end{aligned}$$

so for the adjoint version we set

$$\begin{aligned}\overline{S}_{N-1} &= \mathbb{E}\left[\, P(S_N)\,\frac{\partial \log p_s}{\partial S_{N-1}} \,|\, S_{N-1}\right] \\ \overline{\theta} &= \mathbb{E}\left[\, P(S_N)\,\frac{\partial \log p_s}{\partial\theta} \,|\, S_{N-1}\right]\end{aligned}$$

and then continue backwards down the path in the usual way.

Second order Greeks

AD methods can be used to compute second order Greeks, but this hits problems with the lack of payoff smoothness – how many payoffs have a continuous first derivative?

My personal preference:

- use pathwise sensitivity analysis (either standard or adjoint) for first order Greeks
- apply bumping to this for second order Greeks
- loss of accuracy from bumping is acceptable – don't see as much accuracy for second order Greeks
- computational complexity is also OK – can't do much better

Early exercise

Bermudan and American options have an optimal exercise boundary.

Optimality means the value is insensitive to small perturbations in the exercise boundary – hence can "freeze" the boundary when computing first order Greeks

Computational procedure:

- use Longstaff-Schwartz regression method to compute an estimate of the continuation value which gives the exercise boundary
- use a second set of paths to estimate price and first order Greeks

Further reading

M.B. Giles and P. Glasserman. 'Smoking adjoints: fast Monte Carlo Greeks', RISK, 19(1):88-92, 2006
— *our original RISK paper*

M. Leclerc, Q. Liang, I. Schneider. 'Fast Monte Carlo Bermudan Greeks', RISK, 22(7):84-88, 2009.

L. Capriotti and M.B. Giles. 'Fast correlation Greeks by adjoint algorithmic differentiation', RISK, 23(5):77-83, 2010
— *correlation Greeks and binning*

L. Capriotti, J. Lee, M. Peacock. 'Real Time Counterparty Credit Risk Management in Monte Carlo', RISK 24(6):86-90, 2011

L. Capriotti and M.B. Giles. 'Adjoint Greeks made easy', RISK, 25(9):96-102, 2012
— *use of AD*

Further reading

M.B. Giles. 'Monte Carlo evaluation of sensitivities in computational finance'. Numerical Analysis report NA-07/12, 2007.
— *use of AD, and introduction of Vibrato idea*

M.B. Giles. 'Vibrato Monte Carlo sensitivities'. In Monte Carlo and Quasi-Monte Carlo Methods 2008, Springer, 2009.
— *Vibrato Monte Carlo for discontinuous payoffs*

C. Kaebe, J.H. Maruhn and E.W. Sachs. 'Adjoint-based Monte Carlo calibration of financial market models'. *Finance and Stochastics*, 13(3):351-379, 2009.
— *adjoint Monte Carlo sensitivities and calibration*

L. Capriotti. 'Fast Greeks by algorithmic differentiation', *Journal of Computational Finance*, 14(3):3-35, 2011.

Adjoint methods in computational finance

Mike Giles

Mathematical and Computational Finance Group,
Mathematical Institute, University of Oxford

Oxford-Man Institute of Quantitative Finance

Global Derivatives USA

Nov 22, 2013

PDEs and finite difference methods

- formulation of adjoint PDEs and finite difference methods
- financial application
- vanilla pricing calculation
- sensitivities for linear explicit discretisations
- nonlinear implicit discretisations

Forward and reverse PDEs

Suppose we are interested in the forward PDE

$$\frac{\partial p}{\partial t} = L_t\, p,$$

where L_t is a spatial operator, subject to Dirac initial data $p(x,0) = \delta(x{-}x_0)$, and we want the value of the output functional

$$(p(\cdot, T),\, f) \equiv \int p(x,T)\, f(x)\, \mathrm{d}x.$$

The adjoint spatial operator L_t^* is defined by the identity

$$(L_t v, w) = (v, L_t^* w), \quad \forall v, w$$

assuming certain homogeneous b.c.'s.

Forward and reverse PDEs

If $u(x,t)$ is the solution of the adjoint PDE

$$\frac{\partial u}{\partial t} = -L_t^*\, u,$$

subject to "initial" data $u(x,T) = f(x)$ then

$$\begin{aligned}
(p(\cdot,T),\, u(\cdot,T)) - (p(\cdot,0),\, u(\cdot,0)) &= \int_0^T \frac{\partial}{\partial t}(p,\, u)\ \mathrm{d}t \\
&= \int_0^T \left(\frac{\partial p}{\partial t},\, u\right) + \left(p,\, \frac{\partial u}{\partial t}\right)\ \mathrm{d}t \\
&= \int_0^T (L_t p,\, u) - (p,\, L_t^* u)\ \mathrm{d}t \\
&= 0,
\end{aligned}$$

and hence $u(x_0, 0) = (p(\cdot, T), f)$.

Forward and reverse PDEs

Hence, to compute our output of interest, we have a choice:

- forward:
  - start with Dirac initial data for $p(x,0)$
  - solve forward PDE for $p(x,t)$
  - compute $(p(\cdot,T),f)$
- reverse:
  - start with "initial" data for $u(x,T)$
  - solve backward PDE for $u(x,t)$
  - output is $u(x_0,0)$

We get the same answer either way, so can choose based on other considerations, such as computational efficiency

Financial relevance

Fokker-Planck (or forward Kolmogorov) equation:

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}(a\,p) = \frac{1}{2}\frac{\partial^2}{\partial x^2}(b^2\,p)$$

for probability density $p(x,t)$ for path S_t satisfying the SDE

$$\mathrm{d}S_t = a(S_t,t)\,\mathrm{d}t + b(S_t,t)\,\mathrm{d}W_t.$$

Feynman-Kac equation:

$$\frac{\partial u}{\partial t} + a\,\frac{\partial u}{\partial x} + \frac{1}{2}\,b^2\,\frac{\partial^2 u}{\partial x^2} = 0$$

where $u(x,t) = \mathbb{E}[f(S_T)|S_t = x]$

Financial relevance

The spatial operators are

$$L\,p \equiv -\frac{\partial}{\partial x}(a\,p) + \frac{1}{2}\frac{\partial^2}{\partial x^2}(b^2\,p)$$

and

$$L^*\,u \equiv a\,\frac{\partial u}{\partial x} + \frac{1}{2}\,b^2\,\frac{\partial^2 u}{\partial x^2}$$

The identity

$$(Lv,w) = (v,L^*w), \quad \forall v,w$$

can be verified by integration by parts, assuming

$a\,v\,w,\quad b^2 v\,\dfrac{\partial w}{\partial x},\quad b^2\dfrac{\partial v}{\partial x}\,w$ are zero on boundary.

Forward and reverse FDEs

Suppose that a numerical finite difference discretisation of the forward problem gives the discrete equivalent

$$p_{n+1} = A_n\,p_n$$

where p_n is an vector of approximations to $p(x_j,t_n)$ at points x_j at time t_n, and A_n is a square matrix.

For example,

$$p_{j,n+1} = p_{j,n} + \frac{\Delta t}{\Delta x^2}\left(p_{j+1,n} - 2p_{j,n} + p_{j-1,n}\right)$$

is an approximation to

$$\frac{\partial p}{\partial t} = \frac{\partial^2 p}{\partial x^2}$$

Forward and reverse FDEs

If there are N timesteps, the output $(p(x,T), f)$ can be approximated as

$$\sum_j p_{j,N}\; f_j\; \Delta x$$

or more generally as $f^T M\, p_N$ where M is a symmetric "mass" matrix, usually either diagonal or tri-diagonal.

The output then has the form

$$f^T M\, p_N = f^T\; M\, A_{N-1}\, A_{N-2}\, \ldots\, A_0\, p_0.$$

Forward and reverse FDEs

Taking the transpose, this can be re-expressed as

$$p_0^T\, v_0$$

where

$$v_0 = A_0^T\; \ldots\; A_{N-2}^T\, A_{N-1}^T\; M\; f$$

The adjoint solution v_n is therefore defined by

$$v_n = A_n^T\; v_{n+1}$$

subject to "initial" data $v_N = M\, f$.

Forward and reverse FDEs

It is often more appropriate to work with

$$u_n = M^{-1} v_n,$$

in which case we have

$$u_n = (M\, A_n^T\, M^{-1})\, u_{n+1}$$

subject to "initial" data

$$u_N = f,$$

and the output functional is $p_0^T\; M\, u_0$.

This is more appropriate because now u_n is an approximation to the adjoint PDE solution $u(x, t_n)$

Financial relevance

In finance, the discrete equations are usually formulated for backward equation:

$$u_n = B_n\, u_{n+1}$$

subject to payoff data $u_N = f$, and the output is $e^T u_0$ where e is a unit vector with a single non-zero entry.

The equivalent discrete adjoint problem is

$$P_{n+1} = B_n^T\, P_n$$

subject to initial data $P_0 = e$, and the output is $P_N^T\, f$.

When there is no discounting (so no $r\,u$ term in Black-Scholes PDE) then P_n corresponds to a vector of discrete probabilities – need to divide by grid spacing to get approximation to probability density.

Financial relevance

With implicit time-marching, we have an equation like

$$A_n\, u_n = C_n\, u_{n+1}$$

so

$$B_n \equiv A_n^{-1} C_n$$

In this case,

$$B_n^T \equiv C_n^T (A_n^T)^{-1}$$

so

$$P_{n+1} = C_n^T (A_n^T)^{-1} P_n$$

Note order reversal: multiplication by C_n and then by A_n^{-1} turns into multiplication by $(A_n^T)^{-1}$ and then by C_n^T

Financial relevance

Which is better – forward or reverse?

- reverse is only possibility for American options, and also gives Delta and Gamma approximations for free
- forward is best for pricing multiple European options
  - for different strikes, a single forward calculation and then a separate vector dot product for each option
  - for different maturities, do a single calculation to the final maturity, and use intermediate values at intermediate maturities
  - particularly useful when calibrating a model to vanilla options?

FDE sensitivities

Suppose we want to compute output $e^T u_0$ where $u_N = f$ and

$$u_n = B_n\, u_{n+1}.$$

Now suppose that f and B_n depend on some parameter θ, and we want to compute the sensitivity to θ.

Standard "forward mode" sensitivity analysis gives sensitivity $e^T \dot{u}_0$ where $\dot{u}_N = \dot{f}$ and

$$\dot{u}_n = B_n\, \dot{u}_{n+1} + \dot{b}_n$$

with

$$\dot{b}_n \equiv \dot{B}_n\, u_{n+1}$$

FDE sensitivities

What is reverse mode adjoint?

Work "backwards":

$$\overline{u}_0 = e$$

$$\overline{u}_{n+1} = B_n^T\, \overline{u}_n, \quad \overline{b}_n = \overline{u}_n$$

$$\overline{f} = \overline{u}_N$$

Note: the original code goes from $n\!=\!N$ to $n\!=\!0$, so the reverse mode goes from $n\!=\!0$ to $n\!=\!N$, using stored values for u_{n+1}.

FDE sensitivities

This gives $\overline{f}$ and $\overline{b}_n$ and then payoff sensitivity is given by

$$\overline{\theta} = \overline{f}^T \dot{f} + \sum_n \overline{b}_n^T \dot{b}_n$$

This can be evaluated using AD software, or hand-coded following the AD algorithm.

$\theta, u_{n+1} \longrightarrow B_n\, u_{n+1}$	original code
$\theta, u_{n+1} \longrightarrow \dot{B}_n\, u_{n+1}$	forward mode, keeping u_{n+1} fixed
$\theta, u_{n+1}, \overline{b}_n \longrightarrow \overline{\theta}$ incr	reverse mode, keeping u_{n+1} fixed

FDE sensitivities

We now consider nonlinear discretisations (e.g. for American options)

In 1D, these are usually one of the following types:

- explicit:

$$u_{j,n} = g(u_{j-1,n+1}, u_{j,n+1}, u_{j+1,n+1})$$

  – function of the nearest "old" values from the previous timestep

- one-step implicit:

$$a_j\, u_{j-1,n} + b_j\, u_{j,n} + c_j\, u_{j+1,n} = g(u_{j-1,n+1}, u_{j,n+1}, u_{j+1,n+1})$$

  – needs solution of tridiagonal system of equations at each timestep

- iterative implicit:

$$g(u_{j-1,n}, u_{j,n}, u_{j+1,n},\; u_{j-1,n+1}, u_{j,n+1}, u_{j+1,n+1}) = 0$$

  – a nonlinear system of simultaneous equations to be solved iteratively

FDE sensitivities

Considering perturbations to these, "forward mode" sensitivity analysis gives

$$A\, \dot{u}_n = C_n\, \dot{u}_{n+1} + \dot{b}_n$$

with tridiagonal A, C and vector $\dot{b}_n$.

For example, in the third case we have $\dot{b}_{j,n} \equiv \dfrac{\partial g}{\partial \theta}$ and

$$A_{j,j-1} \equiv -\frac{\partial g}{\partial u_{j-1,n}}, \quad A_{j,j} \equiv -\frac{\partial g}{\partial u_{j,n}}, \quad A_{j,j+1} \equiv -\frac{\partial g}{\partial u_{j+1,n}},$$

$$C_{j,j-1} \equiv \frac{\partial g}{\partial u_{j-1,n}}, \quad C_{j,j} \equiv \frac{\partial g}{\partial u_{j,n}}, \quad C_{j,j+1} \equiv \frac{\partial g}{\partial u_{j+1,n}},$$

with $A, C, \dot{b}_n$ dependent on $u_{j-1,n}, u_{j,n}, u_{j+1,n},\; u_{j-1,n+1}, u_{j,n+1}, u_{j+1,n+1}$.

FDE sensitivities

"Reverse mode" gives

$$\overline{u}_{n+1} = C_n^T\, (A_n^T)^{-1} \overline{u}_n, \quad \overline{b}_n = (A_n^T)^{-1}\, \overline{u}_n$$

This again gives $\overline{b}_n$ and AD ideas can then be used to compute the increments to $\overline{\theta}$.

So far, I have talked of θ being a single input parameter, but it can be a vector of input parameters.

The key is that they all use the same $\overline{f}$ and $\overline{b}_n$, and it is just this final AD step which depends on θ, and the cost is independent of the number of parameters.

Adjoint methods in computational finance

Mike Giles

Mathematical and Computational Finance Group,
Mathematical Institute, University of Oxford

Oxford-Man Institute of Quantitative Finance

Global Derivatives USA

Nov 22, 2013

PDEs and finite difference methods

- what can go wrong?
- calibration using Fokker-Planck discretisation
- local volatility example

What can go wrong?

Differentiation like this gives the sensitivity of the numerical approximation to changes in the input parameters.

This is not necessarily a good approximation to the true sensitivity

Simplest example: a digital put option with strike K when wanting to compute $\dfrac{\partial V}{\partial K}$, the sensitivity of the option price to the strike

What can go wrong?

Using the simplest numerical approximation,

$$f_j = H(K - S_j)$$

we then get $\dot{f} = 0$ which leads to a zero sensitivity!

Using a better approximation

$$f_j = \frac{1}{\Delta S} \int_{S_j - \frac{1}{2}\Delta S}^{S_j + \frac{1}{2}\Delta S} H(K - S)\, \mathrm{d}S$$

gives an $O(\Delta S^2)$ approximation to the price, and an $O(\Delta S)$ approximation to the sensitivity to K.

What can go wrong?

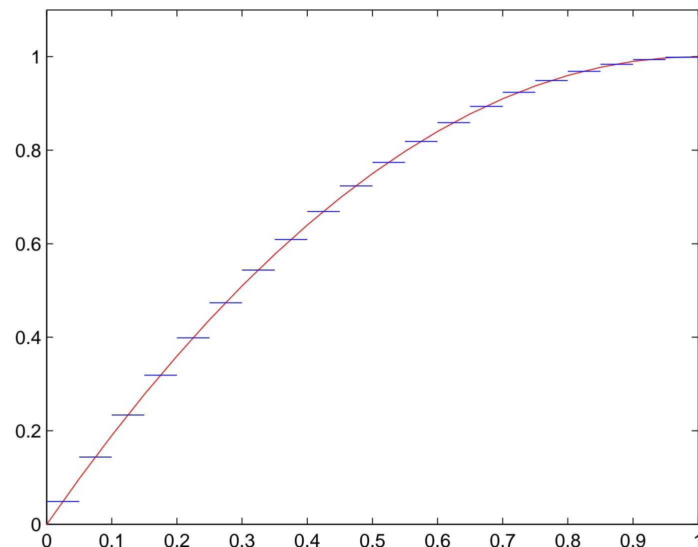


Figure: A stepped approximation to the function $2x - x^2$

What can go wrong?

More generally, discontinuities are not the only problem.

Suppose our analytic problem with input x has solution

$$u = x^2$$

and our discrete approximation with step size $h \ll 1$ is

$$u_h = x^2 + h^2 \sin(x/h)$$

then $u_h - u = O(h^2)$ but $u_h' - u' = O(h)$

This seems to be typical, that in bad cases you lose one order of convergence each time you differentiate.

What can go wrong?

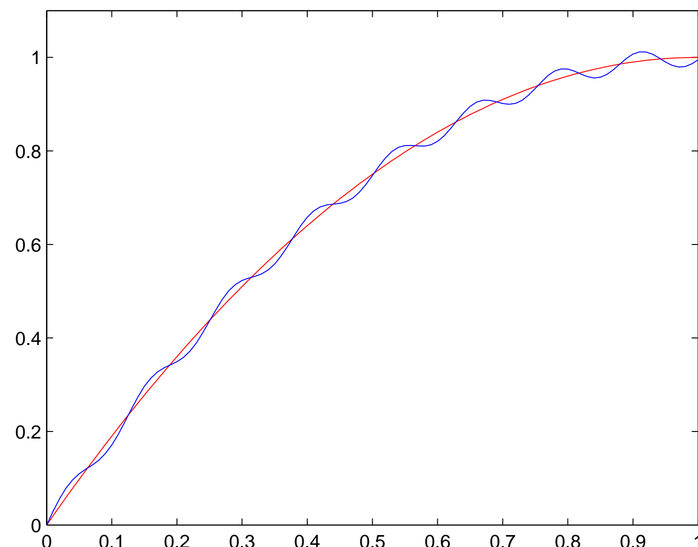


Figure: A wavy approximation to the function $2x - x^2$

What can go wrong?

Careful construction of the approximation can usually avoid these problems.

In the digital put case, the problem was the strike moving across the grid.

Solution: move the grid with the strike at maturity $t = T$, keeping the end at time $t = 0$ fixed.

$$\log S_j(t) = \log S_j^{(0)} + (\log K - \log K^{(0)}) \frac{t}{T}$$

This uses a baseline grid $S_j^{(0)}$ corresponding to the true strike $K^{(0)}$ then considers perturbations to this which move with the strike.

Use of adjoint sensitivities

Fokker-Planck discretisation:

- standard calculation goes forward in time, then performs a separate vector dot product for each vanilla European option
- adjoint sensitivity calculation goes backward in time, gives sensitivity of vanilla prices to initial prices, model constants
- if the Greeks are needed for each option, then a separate adjoint calculation is needed for each – might be better to use "forward mode" AD instead, depending on number of parameters and options
- one adjoint calculation can give a weighted average of Greeks – useful for calibrating a model to market data

Use of adjoint sensitivities

A calibration procedure might find the optimum vector of parameters θ which minimises the mean square difference between vanilla option model prices and market prices:

$$\frac{1}{2} \sum_k \left(C^{(k)}_{model}(\theta) - C^{(k)}_{market} \right)^2$$

Gradient-based optimisation would need to compute

$$\sum_k \left(C^{(k)}_{model} - C^{(k)}_{market} \right) \frac{\partial C^{(k)}_{model}}{\partial \theta}$$

which is just a weighted average (with both positive and negative weights) of the Greeks.

Use of adjoint sensitivities

Since the vanilla option price is of the form

$$C^{(k)}_{model} = f_k^T P_N$$

then, provided f_k does not depend on θ, the adjoint calculation works backwards in time from the "initial" condition:

$$\overline{P}_N = \sum_k \left(C^{(k)}_{model} - C^{(k)}_{market)} \right) f_k$$

Use of adjoint sensitivities

Black-Scholes discretisation:

- standard calculation goes backward in time for pricing an exotic option, with possible path-dependency and optional exercise
- adjoint sensitivity calculation goes forward in time, giving sensitivity of price to initial prices, model constants, etc.

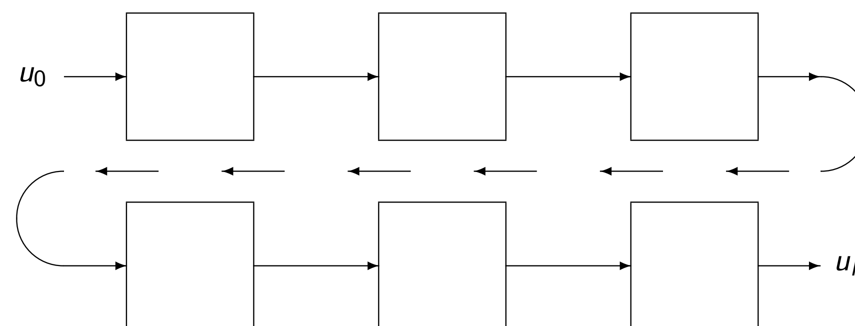
Use of adjoint sensitivities

Many applications may involve a process which goes through several stages:

- market implied vol $\sigma_I \implies$ local vol σ_L at a few points using Dupire's formula
- local vol σ_L at a few points $\implies \sigma_L, \sigma_L'$ through cubic spline construction
- $\sigma_L, \sigma_L' \implies \sigma$ at FD grid points using cubic spline interpolation
- σ at FD grid points $\implies$ option value V using FD calculation

Generic black-box problem

Remember generic black-box viewpoint



Key assumption: each step is (locally) differentiable

Generic black-box problem

Forward mode:

$$\dot{u}_{n+1} = D_n\ \dot{u}_n, \qquad D_n \equiv \frac{\partial u_{n+1}}{\partial u_n}$$

Reverse mode:

$$\overline{u}_n = D_n^T\ \overline{u}_{n+1}$$

starting from given $\overline{u}_N$, and with all of the D_n or u_n stored from the original black-box computation.

Validation:

$$\frac{\partial u_N}{\partial u_n}\ \frac{\partial u_n}{\partial \theta} = \frac{\partial u_N}{\partial u_{n+1}}\ \frac{\partial u_{n+1}}{\partial \theta} \quad \Longrightarrow \quad \overline{u}_n^T\ \dot{u}_n = \overline{u}_{n+1}^T\ \dot{u}_{n+1}$$

This must hold for any $\dot{u}_n, \overline{u}_{n+1}$ – very helpful for checking the forward and reverse mode versions of each black-box component.

Use of adjoint sensitivities

To obtain the sensitivity of the option value to changes in the market implied vol, go through all of the stages in the reverse order:

- $\overline{V} \implies \overline{\sigma}$
- $\overline{\sigma} \implies \overline{\sigma_L}, \overline{\sigma_L'}$
- $\overline{\sigma_L}, \overline{\sigma_L'} \implies \overline{\sigma_L}$
- $\overline{\sigma_L} \implies \overline{\sigma_I}$

Each stage needs to be developed and validated separately, then they all fit together in a modular way.

Use of adjoint sensitivities

It is not necessary to use adjoint techniques at each stage.

For example, the final stage in the last example computes

$$\overline{\sigma_I} = \left(\frac{\partial \sigma_L}{\partial \sigma_I}\right)^T \overline{\sigma_L}$$

The matrix

$$\frac{\partial \sigma_L}{\partial \sigma_I}$$

can be obtained by forward mode sensitivity analysis (more expensive), or approximated by bumping (more expensive and less accurate)

Cubic spline step

For a point $S_j < S < S_{j+1}$, cubic spline interpolation is defined by an equation of the form

$$\sigma(S) = a_j(S)\ \sigma_j + b_j(S)\ \sigma_{j+1} + c_j(S)\ \sigma'_j + d_j(S)\ \sigma'_{j+1},$$

where $a_j(S), b_j(S), c_j(S), d_j(S)$ are cubic polynomials.

The σ' values are obtained from the σ values by solving a tri-diagonal system of equations:

$$A\ \sigma' = B\ \sigma$$

Cubic spline step

In the forward mode we get

$$A\ \dot{\sigma}' = B\ \dot{\sigma},$$

and then

$$\dot{\sigma}(S) = a_j(S)\ \dot{\sigma}_j + b_j(S)\ \dot{\sigma}_{j+1} + c_j(S)\ \dot{\sigma}'_j + d_j(S)\ \dot{\sigma}'_{j+1}$$

assuming that the point at which the spline is evaluated does not change.

As usual, this is relatively intuitive.

Cubic spline step

In the reverse mode we have

$$\begin{aligned} \overline{\sigma}_j &\mathrel{+}= a_j(S)\ \overline{\sigma}(S) \\ \overline{\sigma}_{j+1} &\mathrel{+}= b_j(S)\ \overline{\sigma}(S) \\ \overline{\sigma'}_j &\mathrel{+}= c_j(S)\ \overline{\sigma}(S) \\ \overline{\sigma'}_{j+1} &\mathrel{+}= d_j(S)\ \overline{\sigma}(S) \end{aligned}$$

which gives the increments to $\overline{\sigma}_j, \overline{\sigma}_{j+1}, \overline{\sigma'}_j, \overline{\sigma'}_{j+1}$ due to the spline evaluation.

Reversing the calculation of the spline derivatives then gives

$$\overline{\sigma} \mathrel{+}= B^T (A^T)^{-1}\ \overline{\sigma'},$$

which adds to $\overline{\sigma}$ the extra dependence due to the way in which σ' is calculated from σ.

Final comments

- for pricing multiple European options, cheaper to solve one Fokker-Planck equation for evolution of density, rather than multiple Black-Scholes equations for option value
- doesn't work for American or Bermudan options because they're nonlinear
- for sensitivity calculations, the big benefit from adjoint methods comes (as usual) when there are lots of sensitivities to be computed – local volatility case is a good example
- must remember there's a potential loss of accuracy when differentiating – a good approximation to the option value does not necessarily imply a good approximation to the Greeks

Further reading

M.B. Giles 'On the iterative solution of adjoint equations', pp.145-152 in Automatic Differentiation: From Simulation to Optimization, G. Corliss, C. Faure, A. Griewank, L. Hascoet, U. Naumann, editors, Springer-Verlag, 2001.
— *adjoint treatment of time-marching and fixed point iteration*

M.B. Giles. 'Collected matrix derivative results for forward and reverse mode algorithmic differentiation'. In Advances in Automatic Differentiation, Springer, 2008.

M.B. Giles. 'An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation'. Numerical Analysis report NA-08/01, 2008.
— *two papers on adjoint linear algebra, second has MATLAB code and tips on code development and validation*

Conclusions

- adjoints can be very efficient for option pricing, calibration and sensitivity analysis
- same result as "standard" approach but a much lower computational cost
- basic elements of discrete adjoint analysis are very simple, although real applications can get quite complex
- automatic differentiation ideas are very important, even if you don't use AD software